Study guide: Finite difference methods for vibration problems

Hans Petter Langtangen[1,2]

[1]Center for Biomedical Computing, Simula Research Laboratory
[2]Department of Informatics, University of Oslo

Aug 11, 2015

A simple vibration problem

$$u''(t) + \omega^2 u = 0, \quad u(0) = I,\ u'(0) = 0,\ t \in (0, T]$$

Exact solution:

$$u(t) = I\cos(\omega t)$$

$u(t)$ oscillates with constant amplitude I and (angular) frequency ω. Period: $P = 2\pi/\omega$.

A centered finite difference scheme; step 1 and 2

- Strategy: follow the four steps of the finite difference method.
- Step 1: Introduce a time mesh, here uniform on $[0, T]$: $t_n = n\Delta t$
- Step 2: Let the ODE be satisfied at each mesh point:

$$u''(t_n) + \omega^2 u(t_n) = 0, \quad n = 1, \ldots, N_t$$

A centered finite difference scheme; step 3

Step 3: Approximate derivative(s) by finite difference approximation(s). Very common (standard!) formula for u'':

$$u''(t_n) \approx \frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2}$$

Use this discrete initial condition together with the ODE at $t = 0$ to eliminate u^{-1}:

$$\frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} = -\omega^2 u^n$$

A centered finite difference scheme; step 4

Step 4: Formulate the computational algorithm. Assume u^{n-1} and u^n are known, solve for unknown u^{n+1}:

$$u^{n+1} = 2u^n - u^{n-1} - \Delta t^2 \omega^2 u^n$$

Nick names for this scheme: Störmer's method or Verlet integration.

Computing the first step

- The formula breaks down for u^1 because u^{-1} is unknown and outside the mesh!
- And: we have not used the initial condition $u'(0) = 0$.

Discretize $u'(0) = 0$ by a centered difference

$$\frac{u^1 - u^{-1}}{2\Delta t} = 0 \quad \Rightarrow \quad u^{-1} = u^1$$

Inserted in the scheme for $n = 0$ gives

$$u^1 = u^0 - \frac{1}{2}\Delta t^2 \omega^2 u^0$$

The computational algorithm

1. $u^0 = I$
2. compute u^1
3. for $n = 1, 2, \ldots, N_t - 1$:
   (a) compute u^{n+1}

More precisely expressed in Python:

```
t = linspace(0, T, Nt+1)   # mesh points in time
dt = t[1] - t[0]           # constant time step.
u = zeros(Nt+1)            # solution

u[0] = I
u[1] = u[0] - 0.5*dt**2*w**2*u[0]
for n in range(1, Nt):
    u[n+1] = 2*u[n] - u[n-1] - dt**2*w**2*u[n]
```

Note: `w` is consistently used for ω in my code.

Operator notation; ODE

With $[D_tD_tu]^n$ as the finite difference approximation to $u''(t_n)$ we can write

$$[D_tD_tu + \omega^2 u = 0]^n$$

$[D_tD_tu]^n$ means applying a central difference with step $\Delta t/2$ twice:

$$[D_t(D_tu)]^n = \frac{[D_tu]^{n+\frac{1}{2}} - [D_tu]^{n-\frac{1}{2}}}{\Delta t}$$

which is written out as

$$\frac{1}{\Delta t}\left(\frac{u^{n+1} - u^n}{\Delta t} - \frac{u^n - u^{n-1}}{\Delta t}\right) = \frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2}\,.$$

Operator notation; initial condition

$$[u = I]^0, \quad [D_{2t}u = 0]^0$$

where $[D_{2t}u]^n$ is defined as

$$[D_{2t}u]^n = \frac{u^{n+1} - u^{n-1}}{2\Delta t}\,.$$

Computing u'

u is often displacement/position, u' is velocity and can be computed by

$$u'(t_n) \approx \frac{u^{n+1} - u^{n-1}}{2\Delta t} = [D_{2t}u]^n$$

Implementation

Core algorithm

```
from numpy import *
from matplotlib.pyplot import *

def solver(I, w, dt, T):
    """
    Solve u'' + w**2*u = 0 for t in (0,T], u(0)=I and u'(0)=0,
    by a central finite difference method with time step dt.
    """
    dt = float(dt)
    Nt = int(round(T/dt))
    u = zeros(Nt+1)
    t = linspace(0, Nt*dt, Nt+1)

    u[0] = I
    u[1] = u[0] - 0.5*dt**2*w**2*u[0]
    for n in range(1, Nt):
        u[n+1] = 2*u[n] - u[n-1] - dt**2*w**2*u[n]
    return u, t
```

Plotting

```
def u_exact(t, I, w):
    return I*cos(w*t)

def visualize(u, t, I, w):
    plot(t, u, 'r--o')
    t_fine = linspace(0, t[-1], 1001)  # very fine mesh for u_e
    u_e = u_exact(t_fine, I, w)
    hold('on')
    plot(t_fine, u_e, 'b-')
    legend(['numerical', 'exact'], loc='upper left')
    xlabel('t')
    ylabel('u')
    dt = t[1] - t[0]
    title('dt=%g' % dt)
    umin = 1.2*u.min();  umax = -umin
    axis([t[0], t[-1], umin, umax])
    savefig('tmp1.png');  savefig('tmp1.pdf')

def test_three_steps():
    I = 1;  w = 2*pi;  dt = 0.1;  T = 1
    u_by_hand = array([1.000000000000000,
                       0.802607911978213,
                       0.288358920740053])
    u, t = solver(I, w, dt, T)
    diff = abs(u_by_hand - u[:3]).max()
    tol = 1E-14
    assert diff < tol

def convergence_rates(m, solver_function, num_periods=8):
    """
    Return m-1 empirical estimates of the convergence rate
    based on m simulations, where the time step is halved
    for each simulation.
    solver_function(I, w, dt, T) solves each problem, where T
    is based on simulation for num_periods periods.
    """
    w = 0.35; I = 0.3       # just chosen values
    P = 2*pi/w              # period
    dt = P/30               # 30 time step per period 2*pi/w
    T = P*num_periods

    dt_values = []
    E_values = []
    for i in range(m):
        u, t = solver_function(I, w, dt, T)
        u_e = u_exact(t, I, w)
        E = sqrt(dt*sum((u_e-u)**2))
        dt_values.append(dt)
        E_values.append(E)
        dt = dt/2

    r = [log(E_values[i-1]/E_values[i])/
         log(dt_values[i-1]/dt_values[i])
         for i in range(1, m, 1)]
    return r
```

```
def test_convergence_rates():
    r = convergence_rates(m=5, solver_function=solver,
        num_periods=8)
    # Accept rate to 1 decimal place
    tol = 0.1
    assert abs(r[-1] - 2.0) < tol

def main(solver_function=solver):
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument('--I', type=float, default=1.0)
    parser.add_argument('--w', type=float, default=2*pi)
    parser.add_argument('--dt', type=float, default=0.05)
    parser.add_argument('--num_periods', type=int, default=5)
    parser.add_argument('--savefig', action='store_true')
    # Hack to allow --SCITOOLS options (read when importing
        scitools.std)
    parser.add_argument('--SCITOOLS_easyviz_backend',
        default='matplotlib')
    a = parser.parse_args()
    I, w, dt, num_periods, savefig = \
       a.I, a.w, a.dt, a.num_periods, a.savefig
    P = 2*pi/w  # one period
    T = P*num_periods
    u, t = solver_function(I, w, dt, T)
    if num_periods <= 10:
        visualize(u, t, I, w)
    else:
        visualize_front(u, t, I, w, savefig)
        #visualize_front_ascii(u, t, I, w)
    #plot_empirical_freq_and_amplitude(u, t, I, w)
    show()

def plot_empirical_freq_and_amplitude(u, t, I, w):
    """
    Find the empirical angular frequency and amplitude of
    simulations in u and t. u and t can be arrays or (in
    the case of multiple simulations) multiple arrays.
    One plot is made for the amplitude and one for the angular
    frequency (just called frequency in the legends).
    """
    from vib_empirical_analysis import minmax, periods,
        amplitudes
    if not isinstance(u, (list,tuple)):
        u = [u]
        t = [t]
    legends1 = []
    legends2 = []
    for i in range(len(u)):
        minima, maxima = minmax(t[i], u[i])
        p = periods(maxima)
        a = amplitudes(minima, maxima)
        figure(1)
        plot(range(len(p)), 2*pi/p)
        legends1.append('frequency, case%d' % (i+1))
        hold('on')
```

```
        figure(2)
        plot(range(len(a)), a)
        hold('on')
        legends2.append('amplitude, case%d' % (i+1))
    figure(1)
    plot(range(len(p)), [w]*len(p), 'k--')
    legends1.append('exact frequency')
    legend(legends1, loc='lower left')
    axis([0, len(a)-1, 0.8*w, 1.2*w])
    savefig('tmp1.png');  savefig('tmp1.pdf')
    figure(2)
    plot(range(len(a)), [I]*len(a), 'k--')
    legends2.append('exact amplitude')
    legend(legends2, loc='lower left')
    axis([0, len(a)-1, 0.8*I, 1.2*I])
    savefig('tmp2.png');  savefig('tmp2.pdf')
    show()


def visualize_front(u, t, I, w, savefig=False, skip_frames=1):
    """
    Visualize u and the exact solution vs t, using a
    moving plot window and continuous drawing of the
    curves as they evolve in time.
    Makes it easy to plot very long time series.
    Plots are saved to files if savefig is True.
    Only each skip_frames-th plot is saved (e.g., if
    skip_frame=10, only each 10th plot is saved to file;
    this is convenient if plot files corresponding to
    different time steps are to be compared).
    """
    import scitools.std as st
    from scitools.MovingPlotWindow import MovingPlotWindow

    # Remove all old plot files tmp_*.png
    import glob, os
    for filename in glob.glob('tmp_*.png'):
        os.remove(filename)

    P = 2*pi/w  # one period
    umin = 1.2*u.min();  umax = -umin
    dt = t[1] - t[0]
    plot_manager = MovingPlotWindow(
        window_width=8*P,
        dt=dt,
        yaxis=[umin, umax],
        mode='continuous drawing')
    frame_counter = 0
    for n in range(1,len(u)):
        if plot_manager.plot(n):
            s = plot_manager.first_index_in_plot
            st.plot(t[s:n+1], u[s:n+1], 'r-1',
                    t[s:n+1], I*cos(w*t)[s:n+1], 'b-1',
                    title='t=%6.3f' % t[n],
                    axis=plot_manager.axis(),
                    show=not savefig) # drop window if savefig
            if savefig and n % skip_frames == 0:
```

```
                filename = 'tmp_%04d.png' % frame_counter
                st.savefig(filename)
                print 'making plot file', filename, 'at t=%g' %
                    t[n]
                frame_counter += 1
        plot_manager.update(n)

def visualize_front_ascii(u, t, I, w, fps=10):
    """
    Plot u and the exact solution vs t line by line in a
    terminal window (only using ascii characters).
    Makes it easy to plot very long time series.
    """
    from scitools.avplotter import Plotter
    import time
    P = 2*pi/w
    umin = 1.2*u.min();  umax = -umin

    p = Plotter(ymin=umin, ymax=umax, width=60, symbols='+o')
    for n in range(len(u)):
        print p.plot(t[n], u[n], I*cos(w*t[n])), \
              '%.1f' % (t[n]/P)
        time.sleep(1/float(fps))

if __name__ == '__main__':
    main()
    raw_input()
```

Main program

```
I = 1
w = 2*pi
dt = 0.05
num_periods = 5
P = 2*pi/w    #  one period
T = P*num_periods
u, t = solver(I, w, dt, T)
visualize(u, t, I, w, dt)
```

User interface: command line

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--I', type=float, default=1.0)
parser.add_argument('--w', type=float, default=2*pi)
parser.add_argument('--dt', type=float, default=0.05)
parser.add_argument('--num_periods', type=int, default=5)
a = parser.parse_args()
I, w, dt, num_periods = a.I, a.w, a.dt, a.num_periods
```

Running the program

vib_undamped.py:

```
Terminal> python vib_undamped.py --dt 0.05 --num_periods 40
```

Generates frames `tmp_vib%04d.png` in files. Can make movie:

```
Terminal> avconv -r 12 -i tmp_vib%04d.png -c:v flv movie.flv
```

Can use `ffmpeg` instead of `avconv`.

Format	Codec and filename
Flash	`-c:v flv movie.flv`
MP4	`-c:v libx264 movie.mp4`
Webm	`-c:v libvpx movie.webm`
Ogg	`-c:v libtheora movie.ogg`

Verification

First steps for testing and debugging

- **Testing very simple solutions**: $u = \text{const}$ or $u = ct + d$ do not apply here (without a force term in the equation: $u'' + \omega^2 u = f$).
- **Hand calculations**: calculate u^1 and u^2 and compare with program.

Checking convergence rates

The next function estimates convergence rates, i.e., it

- performs m simulations with halved time steps: $2^{-k}\Delta t$, $k = 0, \ldots, m-1$,
- computes the L_2 norm of the error, $E = \sqrt{\Delta t_i \sum_{n=0}^{N_t-1}(u^n - u_{\mathrm{e}}(t_n))^2}$ in each case,
- estimates the rates r_i from two consecutive experiments $(\Delta t_{i-1}, E_{i-1})$ and $(\Delta t_i, E_i)$, assuming $E_i = C\Delta t_i^{r_i}$ and $E_{i-1} = C\Delta t_{i-1}^{r_i}$:

Implementational details

```
def convergence_rates(m, solver_function, num_periods=8):
    """
    Return m-1 empirical estimates of the convergence rate
    based on m simulations, where the time step is halved
    for each simulation.
    solver_function(I, w, dt, T) solves each problem, where T
    is based on simulation for num_periods periods.
    """
    w = 0.35; I = 0.3       # just chosen values
    P = 2*pi/w              # period
    dt = P/30               # 30 time step per period 2*pi/w
    T = P*num_periods

    dt_values = []
    E_values = []
    for i in range(m):
        u, t = solver_function(I, w, dt, T)
        u_e = u_exact(t, I, w)
        E = sqrt(dt*sum((u_e-u)**2))
        dt_values.append(dt)
        E_values.append(E)
        dt = dt/2

    r = [log(E_values[i-1]/E_values[i])/
         log(dt_values[i-1]/dt_values[i])
         for i in range(1, m, 1)]
    return r
```

Result: `r` contains values equal to 2.00 - as expected!

Nose test

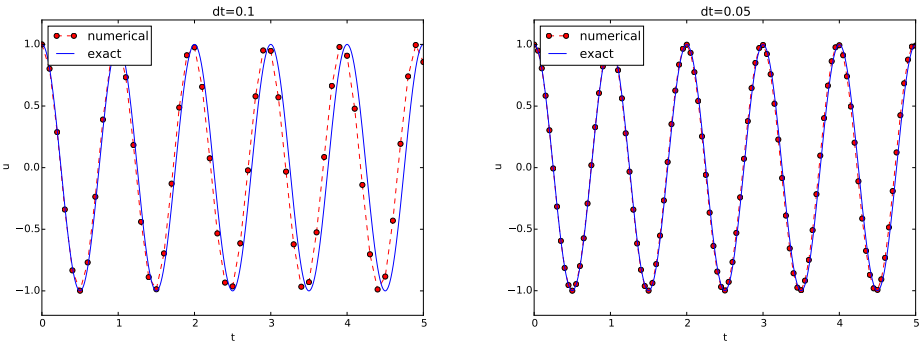
Use final `r[-1]` in a unit test:

```
def test_convergence_rates():
    r = convergence_rates(m=5, solver_function=solver,
        num_periods=8)
    # Accept rate to 1 decimal place
    tol = 0.1
    assert abs(r[-1] - 2.0) < tol
```

Complete code in `vib_undamped.py`.

Long time simulations

Effect of the time step on long simulations



- The numerical solution seems to have right amplitude.
- There is an angular frequency error (reduced by reducing the time step).
- The total angular frequency error seems to grow with time.

Using a moving plot window

- In long time simulations we need a plot window that follows the solution.
- Method 1: `scitools.MovingPlotWindow`.
- Method 2: `scitools.avplotter` (ASCII vertical plotter).

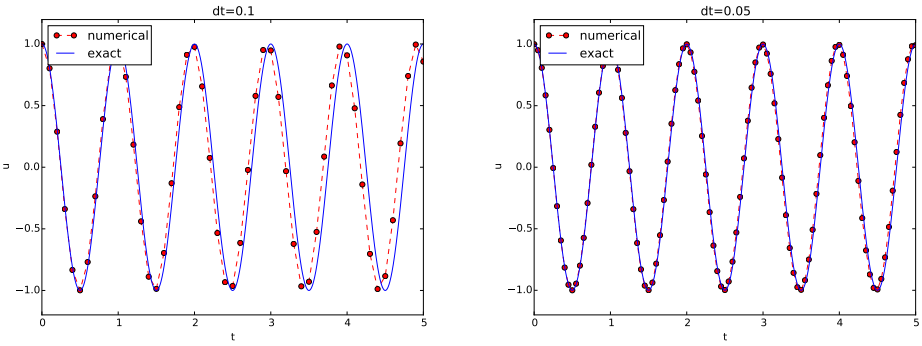
Example:

```
Terminal> python vib_undamped.py --dt 0.05 --num_periods 40
```

Movie of the moving plot window.

Analysis of the numerical scheme

Can we understand the frequency error?



Movie of the angular frequency error

$u'' + \omega^2 u = 0$, $u(0) = 1$, $u'(0) = 0$, $\omega = 2\pi$, $u_{\mathrm{e}}(t) = \cos(2\pi t)$, $\Delta t = 0.05$ (20 intervals per period)

`mov-vib/vib_undamped_movie_dt0.05/movie.ogg`

We can derive an exact solution of the discrete equations

- We have a linear, homogeneous, difference equation for u^n.
- Has solutions $u^n \sim IA^n$, where A is unknown (number).
- Here: $u_{\mathrm{e}}(t) = I\cos(\omega t) \sim I\exp(i\omega t) = I(e^{i\omega\Delta t})^n$
- Trick for simplifying the algebra: $u^n = IA^n$, with $A = \exp(i\tilde{\omega}\Delta t)$, then find $\tilde{\omega}$
- $\tilde{\omega}$: unknown *numerical frequency* (easier to calculate than A)
- $\omega - \tilde{\omega}$ is the angular *frequency error*
- Use the real part as the physical relevant part of a complex expression

Calculations of an exact solution of the discrete equations

$$u^n = IA^n = I\exp(\tilde{\omega}\Delta t\, n) = I\exp(\tilde{\omega}t) = I\cos(\tilde{\omega}t) + iI\sin(\tilde{\omega}t)\,.$$

$$\begin{aligned}
[D_tD_tu]^n &= \frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} \\
&= I\frac{A^{n+1} - 2A^n + A^{n-1}}{\Delta t^2} \\
&= I\frac{\exp(i\tilde{\omega}(t+\Delta t)) - 2\exp(i\tilde{\omega}t) + \exp(i\tilde{\omega}(t-\Delta t))}{\Delta t^2} \\
&= I\exp(i\tilde{\omega}t)\frac{1}{\Delta t^2}\left(\exp(i\tilde{\omega}(\Delta t)) + \exp(i\tilde{\omega}(-\Delta t)) - 2\right) \\
&= I\exp(i\tilde{\omega}t)\frac{2}{\Delta t^2}\left(\cosh(i\tilde{\omega}\Delta t) - 1\right) \\
&= I\exp(i\tilde{\omega}t)\frac{2}{\Delta t^2}\left(\cos(\tilde{\omega}\Delta t) - 1\right) \\
&= -I\exp(i\tilde{\omega}t)\frac{4}{\Delta t^2}\sin^2(\frac{\tilde{\omega}\Delta t}{2})
\end{aligned}$$

Solving for the numerical frequency

The scheme with $u^n = I\exp(i\omega\tilde{\Delta}t\, n)$ inserted gives

$$-I\exp(i\tilde{\omega}t)\frac{4}{\Delta t^2}\sin^2(\frac{\tilde{\omega}\Delta t}{2}) + \omega^2 I\exp(i\tilde{\omega}t) = 0$$

which after dividing by $I\exp(i\tilde{\omega}t)$ results in

$$\frac{4}{\Delta t^2}\sin^2(\frac{\tilde{\omega}\Delta t}{2}) = \omega^2$$

Solve for $\tilde{\omega}$:

$$\tilde{\omega} = \pm\frac{2}{\Delta t}\sin^{-1}\left(\frac{\omega\Delta t}{2}\right)$$

- Frequency error because $\tilde{\omega} \neq \omega$.
- Note: dimensionless number $p = \omega\Delta t$ is the key parameter (i.e., no of time intervals per period is important, not Δt itself)
- But how good is the approximation $\tilde{\omega}$ to ω?

Polynomial approximation of the frequency error

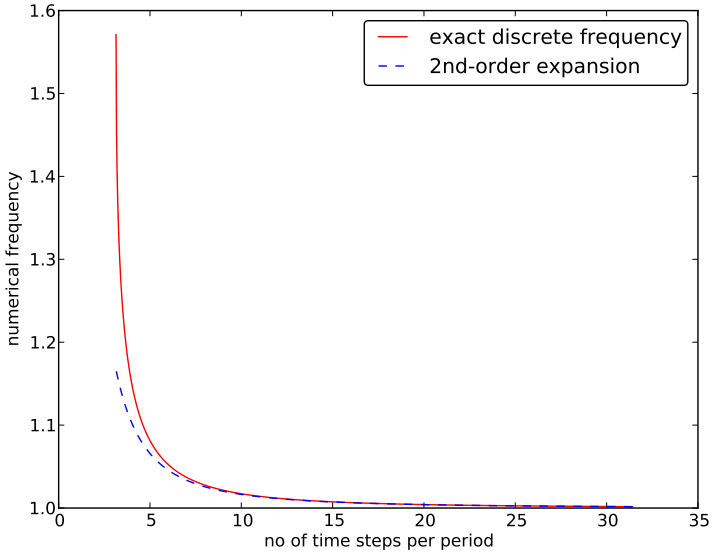
Taylor series expansion for small Δt gives a formula that is easier to understand:

```
>> from sympy import *
>> dt, w = symbols('dt w')
>> w_tilde = asin(w*dt/2).series(dt, 0, 4)*2/dt
>> print w_tilde
(dt*w + dt**3*w**3/24 + O(dt**4))/dt   # note the final "/dt"
```

$$\tilde{\omega} = \omega\left(1 + \frac{1}{24}\omega^2\Delta t^2\right) + \mathcal{O}(\Delta t^3)$$

The numerical frequency is too large (to fast oscillations).

Plot of the frequency error



Recommendation: 25-30 points per period.

Exact discrete solution

$$u^n = I\cos\left(\tilde{\omega}n\Delta t\right), \quad \tilde{\omega} = \frac{2}{\Delta t}\sin^{-1}\left(\frac{\omega\Delta t}{2}\right)$$

The error mesh function,

$$e^n = u_{\mathrm{e}}(t_n) - u^n = I\cos\left(\omega n\Delta t\right) - I\cos\left(\tilde{\omega}n\Delta t\right)$$

is ideal for verification and further analysis!

$$e^n = I\cos\left(\omega n\Delta t\right) - I\cos\left(\tilde{\omega}n\Delta t\right) = -2I\sin\left(t\frac{1}{2}\left(\omega - \tilde{\omega}\right)\right)\sin\left(t\frac{1}{2}\left(\omega + \tilde{\omega}\right)\right)$$

Convergence of the numerical scheme

Can easily show *convergence*:

$$e^n \to 0 \text{ as } \Delta t \to 0,$$

because

$$\lim_{\Delta t\to 0}\tilde{\omega} = \lim_{\Delta t\to 0}\frac{2}{\Delta t}\sin^{-1}\left(\frac{\omega\Delta t}{2}\right) = \omega,$$

by L'Hopital's rule or simply asking `sympy`: or WolframAlpha:

```
>> import sympy as sym
>> dt, w = sym.symbols('x w')
>> sym.limit((2/dt)*sym.asin(w*dt/2), dt, 0, dir='+')
w
```

Stability

Observations:

- Numerical solution has constant amplitude (desired!), but an angular frequency error
- Constant amplitude requires $\sin^{-1}(\omega\Delta t/2)$ to be real-valued $\Rightarrow |\omega\Delta t/2| \leq 1$
- $\sin^{-1}(x)$ is complex if $|x| > 1$, and then $\tilde{\omega}$ becomes complex

What is the consequence of complex $\tilde{\omega}$?

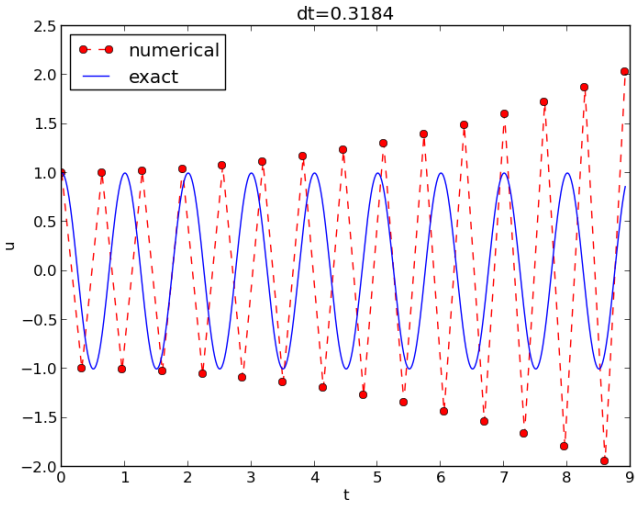
- Set $\tilde{\omega} = \tilde{\omega}_r + i\tilde{\omega}_i$
- Since $\sin^{-1}(x)$ has a *negative* imaginary part for $x > 1$, $\exp(i\omega\tilde{t}) = \exp(-\tilde{\omega}_i t)\exp(i\tilde{\omega}_r t)$ leads to exponential growth $e^{-\tilde{\omega}_i t}$ when $-\tilde{\omega}_i t > 0$
- This is *instability* because the qualitative behavior is wrong

The stability criterion

Cannot tolerate growth and must therefore demand a *stability criterion*

$$\frac{\omega\Delta t}{2} \leq 1 \quad \Rightarrow \quad \Delta t \leq \frac{2}{\omega}$$

Try $\Delta t = \frac{2}{\omega} + 9.01 \cdot 10^{-5}$ (*slightly* too big!):



Summary of the analysis

We can draw three important conclusions:

1. The key parameter in the formulas is $p = \omega\Delta t$ (dimensionless)
   (a) Period of oscillations: $P = 2\pi/\omega$
   (b) Number of time steps per period: $N_P = P/\Delta t$
   (c) $\Rightarrow\ p = \omega\Delta t = 2\pi/N_P \sim 1/N_P$
   (d) The smallest possible N_P is 2 $\Rightarrow p \in (0, \pi]$
2. For $p \leq 2$ the amplitude of u^n is constant (stable solution)
3. u^n has a relative frequency error $\tilde{\omega}/\omega \approx 1 + \frac{1}{24}p^2$, making numerical peaks occur too early

Alternative schemes based on 1st-order equations

Rewriting 2nd-order ODE as system of two 1st-order ODEs

The vast collection of ODE solvers (e.g., in Odespy) cannot be applied to

$$u'' + \omega^2 u = 0$$

unless we write this higher-order ODE as a system of 1st-order ODEs.

Introduce an auxiliary variable $v = u'$:

$$u' = v, \tag{1}$$
$$v' = -\omega^2 u\,. \tag{2}$$

Initial conditions: $u(0) = I$ and $v(0) = 0$.

The Forward Euler scheme

We apply the Forward Euler scheme to each component equation:

$$[D_t^+ u = v]^n,$$
$$[D_t^+ v = -\omega^2 u]^n,$$

or written out,

$$u^{n+1} = u^n + \Delta t v^n, \tag{3}$$
$$v^{n+1} = v^n - \Delta t \omega^2 u^n\,. \tag{4}$$

The Backward Euler scheme

We apply the Backward Euler scheme to each component equation:

$$[D_t^- u = v]^{n+1}, \tag{5}$$
$$[D_t^- v = -\omega u]^{n+1}\,. \tag{6}$$

Written out:

$$u^{n+1} - \Delta t v^{n+1} = u^n, \tag{7}$$
$$v^{n+1} + \Delta t \omega^2 u^{n+1} = v^n\,. \tag{8}$$

This is a *coupled* 2×2 system for the new values at $t = t_{n+1}$!

The Crank-Nicolson scheme

$$[D_t u = \overline{v}^t]^{n+\frac{1}{2}}, \tag{9}$$
$$[D_t v = -\omega \overline{u}^t]^{n+\frac{1}{2}}\,. \tag{10}$$

The result is also a coupled system:

$$u^{n+1} - \frac{1}{2}\Delta t v^{n+1} = u^n + \frac{1}{2}\Delta t v^n, \tag{11}$$
$$v^{n+1} + \frac{1}{2}\Delta t \omega^2 u^{n+1} = v^n - \frac{1}{2}\Delta t \omega^2 u^n\,. \tag{12}$$

Comparison of schemes via Odespy

Can use Odespy to compare many methods for first-order schemes:

```
import odespy
import numpy as np

def f(u, t, w=1):
    u, v = u  # u is array of length 2 holding our [u, v]
    return [v, -w**2*u]

def run_solvers_and_plot(solvers, timesteps_per_period=20,
                         num_periods=1, I=1, w=2*np.pi):
    P = 2*np.pi/w  # duration of one period
    dt = P/timesteps_per_period
    Nt = num_periods*timesteps_per_period
    T = Nt*dt
    t_mesh = np.linspace(0, T, Nt+1)

    legends = []
    for solver in solvers:
        solver.set(f_kwargs={'w': w})
        solver.set_initial_condition([I, 0])
        u, t = solver.solve(t_mesh)
```

Forward and Backward Euler and Crank-Nicolson

```
solvers = [
    odespy.ForwardEuler(f),
    # Implicit methods must use Newton solver to converge
    odespy.BackwardEuler(f, nonlinear_solver='Newton'),
    odespy.CrankNicolson(f, nonlinear_solver='Newton'),
    ]
```

Two plot types:

- $u(t)$ vs t
- Parameterized curve $(u(t), v(t))$ in *phase space*
- Exact curve is an ellipse: $(I\cos\omega t, -\omega I\sin\omega t)$, closed and periodic

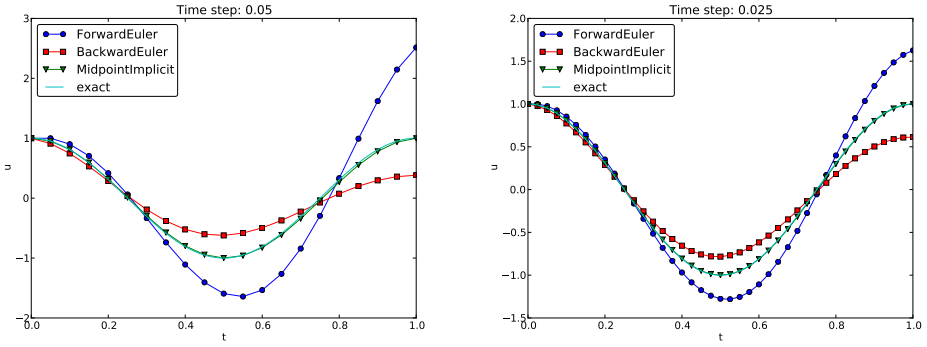
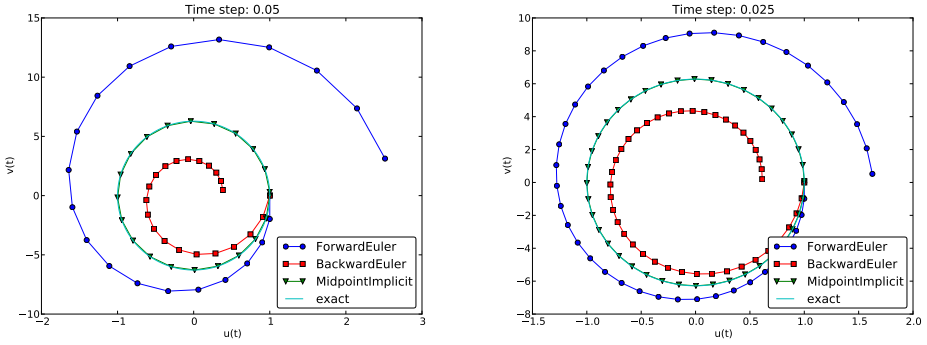


Figure 1: Comparison of classical schemes.

Phase plane plot of the numerical solutions



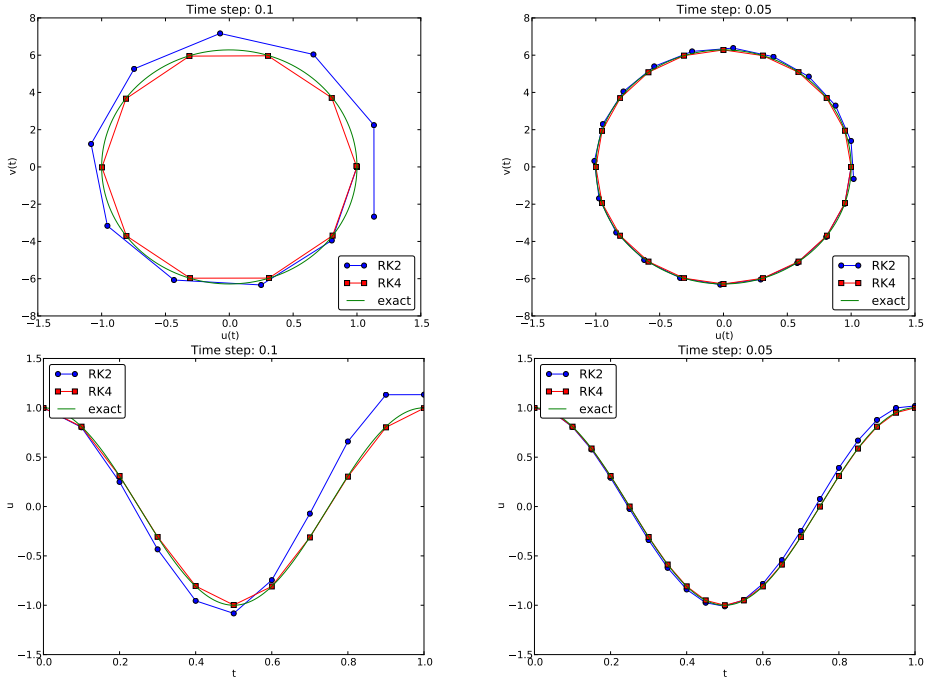
Note: CrankNicolson in Odespy leads to the name MidpointImplicit in plots.

Plain solution curves

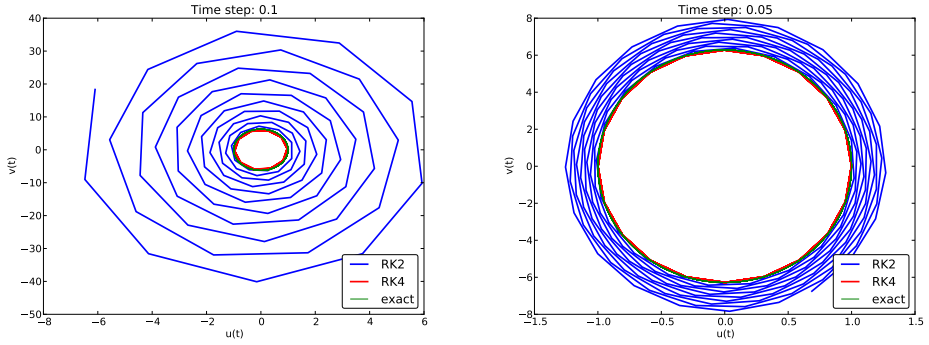
Observations from the figures

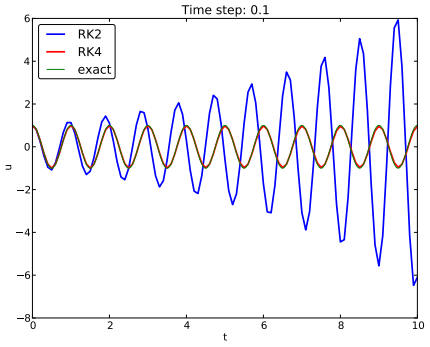
- Forward Euler has growing amplitude and outward (u, v) spiral - pumps energy into the system.
- Backward Euler is opposite: decreasing amplitude, inward sprial, extracts energy.
- **Forward and Backward Euler are useless for vibrations.**
- Crank-Nicolson (MidpointImplicit) looks much better.

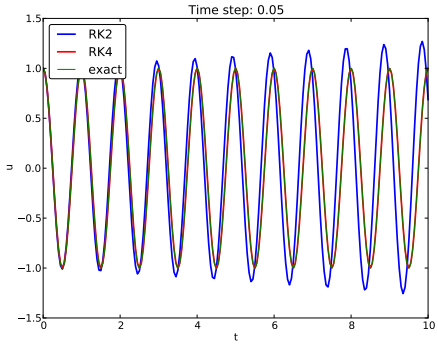
Runge-Kutta methods of order 2 and 4; short time series



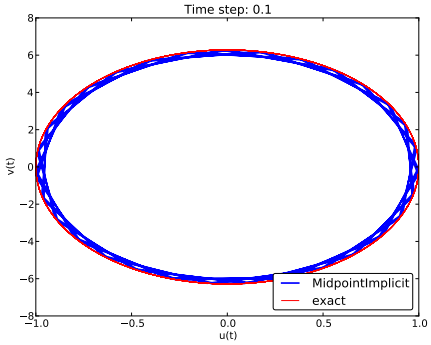
Runge-Kutta methods of order 2 and 4; longer time series

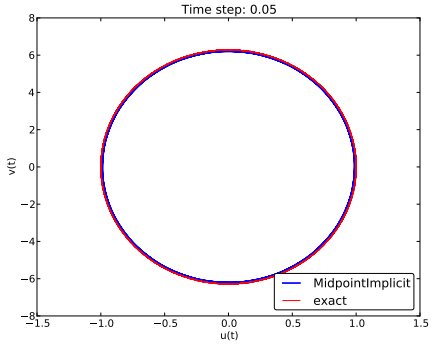


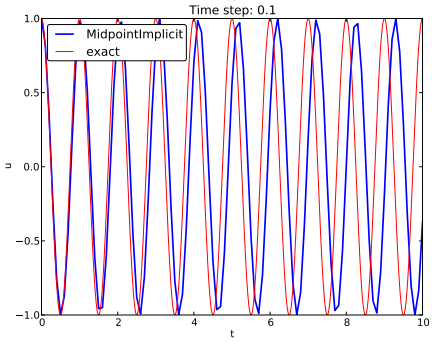


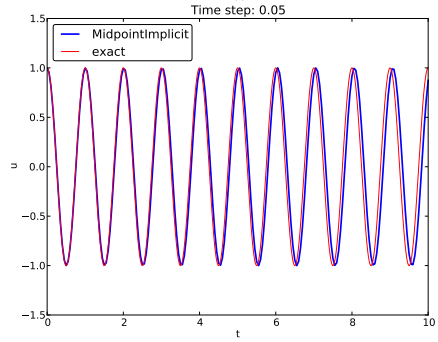


Crank-Nicolson; longer time series









(MidpointImplicit means CrankNicolson in Odespy)

Observations of RK and CN methods

- 4th-order Runge-Kutta is very accurate, also for large Δt.
- 2th-order Runge-Kutta is almost as bad as Forward and Backward Euler.
- Crank-Nicolson is accurate, but the amplitude is not as accurate as the difference scheme for $u'' + \omega^2 u = 0$.

Energy conservation property

The model

$$u'' + \omega^2 u = 0, \quad u(0) = I,\ u'(0) = V,$$

has the nice *energy conservation property* that

$$E(t) = \frac{1}{2}(u')^2 + \frac{1}{2}\omega^2 u^2 = \text{const}\,.$$

This can be used to check solutions.

Derivation of the energy conservation property

Multiply $u'' + \omega^2 u = 0$ by u' and integrate:

$$\int_0^T u''u'dt + \int_0^T \omega^2 uu'dt = 0\,.$$

Observing that

$$u''u' = \frac{d}{dt}\frac{1}{2}(u')^2, \quad uu' = \frac{d}{dt}\frac{1}{2}u^2,$$

we get

$$\int_0^T (\frac{d}{dt}\frac{1}{2}(u')^2 + \frac{d}{dt}\frac{1}{2}\omega^2 u^2)dt = E(T) - E(0),$$

where

$$E(t) = \frac{1}{2}(u')^2 + \frac{1}{2}\omega^2 u^2$$

Remark about $E(t)$

$E(t)$ does not measure energy, energy per mass unit.

Starting with an ODE coming directly from Newton's 2nd law $F = ma$ with a spring force $F = -ku$ and $ma = mu''$ (a: acceleration, u: displacement), we have

$$mu'' + ku = 0$$

Integrating this equation gives a physical energy balance:

$$E(t) = \underbrace{\frac{1}{2}mv^2}_{\text{kinetic energy}} + \underbrace{\frac{1}{2}ku^2}_{\text{potential energy}} = E(0), \quad v = u'$$

Note: the balance is not valid if we add other terms to the ODE.

The Euler-Cromer method; idea

2x2 system for $u'' + \omega^2 u = 0$:

$$v' = -\omega^2 u$$
$$u' = v$$

Forward-backward discretization:

- Update v with Forward Euler
- Update u with Backward Euler, using latest v

$$[D_t^+ v = -\omega^2 u]^n \tag{13}$$
$$[D_t^- u = v]^{n+1} \tag{14}$$

The Euler-Cromer method; complete formulas

Written out:

$$u^0 = I, \tag{15}$$
$$v^0 = 0, \tag{16}$$
$$v^{n+1} = v^n - \Delta t \omega^2 u^n \tag{17}$$
$$u^{n+1} = u^n + \Delta t v^{n+1} \tag{18}$$

Names: Forward-backward scheme, Semi-implicit Euler method, symplectic Euler, semi-explicit Euler, Newton-Stormer-Verlet, and *Euler-Cromer*.

Euler-Cromer is equivalent to the scheme for $u'' + \omega^2 u = 0$

- Forward Euler and Backward Euler have error $\mathcal{O}(\Delta t)$
- What about the overall scheme? Expect $\mathcal{O}(\Delta t)$...

We can eliminate v^n and v^{n+1}, resulting in

$$u^{n+1} = 2u^n - u^{n-1} - \Delta t^2 \omega^2 u^n$$

which is the centered finite differrence scheme for $u'' + \omega^2 u = 0$!

The schemes are not equivalent wrt the initial conditions

$$u' = v = 0 \quad \Rightarrow \quad v^0 = 0,$$

so

$$v^1 = v^0 - \Delta t\omega^2 u^0 = -\Delta t\omega^2 u^0$$
$$u^1 = u^0 + \Delta t v^1 = u^0 - \Delta t\omega^2 u^0 ! = \underbrace{u^0 - \frac{1}{2}\Delta t\omega^2 u^0}_{\text{from } [D_t D_t u + \omega^2 u = 0]^n \text{ and } [D_{2t} u = 0]^0}$$

The exact discrete solution derived earlier does not fit the Euler-Cromer scheme because of mismatch for u^1.

Generalization: damping, nonlinear spring, and external excitation

$$mu'' + f(u') + s(u) = F(t), \quad u(0) = I,\ u'(0) = V,\ t \in (0, T]$$

Input data: m, $f(u')$, $s(u)$, $F(t)$, I, V, and T.

Typical choices of f and s:

- linear damping $f(u') = bu$, or
- quadratic damping $f(u') = bu'|u'|$
- linear spring $s(u) = cu$
- nonlinear spring $s(u) \sim \sin(u)$ (pendulum)

A centered scheme for linear damping

$$[mD_t D_t u + f(D_{2t} u) + s(u) = F]^n$$

Written out

$$m\frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} + f(\frac{u^{n+1} - u^{n-1}}{2\Delta t}) + s(u^n) = F^n$$

Assume $f(u')$ is linear in $u' = v$:

$$u^{n+1} = \left(2mu^n + (\frac{b}{2}\Delta t - m)u^{n-1} + \Delta t^2(F^n - s(u^n))\right)(m + \frac{b}{2}\Delta t)^{-1}$$

Initial conditions

$u(0) = I$, $u'(0) = V$:

$$
\begin{aligned}
[u = I]^0 \quad &\Rightarrow \quad u^0 = I \\
[D_{2t}u = V]^0 \quad &\Rightarrow \quad u^{-1} = u^1 - 2\Delta t V
\end{aligned}
$$

End result:

$$u^1 = u^0 + \Delta t\, V + \frac{\Delta t^2}{2m}(-bV - s(u^0) + F^0)$$

Same formula for u^1 as when using a centered scheme for $u'' + \omega u = 0$.

Linearization via a geometric mean approximation

- $f(u') = bu'|u'|$ leads to a quadratic equation for u^{n+1}
- Instead of solving the quadratic equation, we use a geometric mean approximation

In general, the geometric mean approximation reads

$$(w^2)^n \approx w^{n-\frac{1}{2}} w^{n+\frac{1}{2}}\,.$$

For $|u'|u'$ at t_n:

$$[u'|u'|]^n \approx u'(t_n + \frac{1}{2})|u'(t_n - \frac{1}{2})|\,.$$

For u' at $t_{n\pm 1/2}$ we use centered difference:

$$u'(t_{n+1/2}) \approx [D_t u]^{n+\frac{1}{2}}, \quad u'(t_{n-1/2}) \approx [D_t u]^{n-\frac{1}{2}}$$

A centered scheme for quadratic damping

After some algebra:

$$
\begin{aligned}
u^{n+1} = &\left(m + b|u^n - u^{n-1}|\right)^{-1} \times \\
&\left(2mu^n - mu^{n-1} + bu^n|u^n - u^{n-1}| + \Delta t^2(F^n - s(u^n))\right)
\end{aligned}
$$

Initial condition for quadratic damping

Simply use that $u' = V$ in the scheme when $t = 0$ ($n = 0$):

$$[mD_tD_tu + bV|V| + s(u) = F]^0$$

which gives

$$u^1 = u^0 + \Delta t V + \frac{\Delta t^2}{2m}\left(-bV|V| - s(u^0) + F^0\right)$$

Algorithm

1. $u^0 = I$
2. compute u^1 (formula depends on linear/quadratic damping)
3. for $n = 1, 2, \ldots, N_t - 1$:
   (a) compute u^{n+1} from formula (depends on linear/quadratic damping)

Implementation

```
def solver(I, V, m, b, s, F, dt, T, damping='linear'):
    dt = float(dt); b = float(b); m = float(m) # avoid integer
        div.
    Nt = int(round(T/dt))
    u = zeros(Nt+1)
    t = linspace(0, Nt*dt, Nt+1)

    u[0] = I
    if damping == 'linear':
        u[1] = u[0] + dt*V + dt**2/(2*m)*(-b*V - s(u[0]) +
            F(t[0]))
    elif damping == 'quadratic':
        u[1] = u[0] + dt*V + \
               dt**2/(2*m)*(-b*V*abs(V) - s(u[0]) + F(t[0]))

    for n in range(1, Nt):
        if damping == 'linear':
            u[n+1] = (2*m*u[n] + (b*dt/2 - m)*u[n-1] +
                      dt**2*(F(t[n]) - s(u[n])))/(m + b*dt/2)
        elif damping == 'quadratic':
            u[n+1] = (2*m*u[n] - m*u[n-1] + b*u[n]*abs(u[n] -
                u[n-1])
                      + dt**2*(F(t[n]) - s(u[n])))/\
                      (m + b*abs(u[n] - u[n-1]))
    return u, t
```

Verification

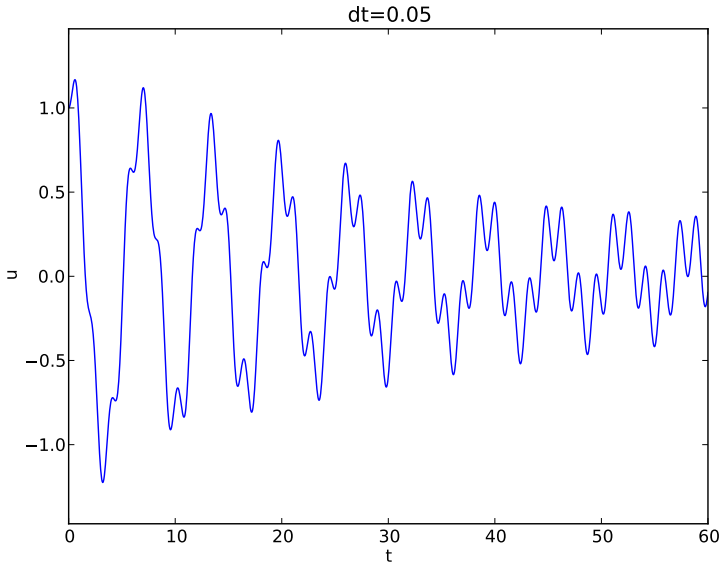
- Constant solution $u_{\mathrm{e}} = I$ ($V = 0$) fulfills the ODE problem and the discrete equations. Ideal for debugging!
- Linear solution $u_{\mathrm{e}} = Vt + I$ fulfills the ODE problem and the discrete equations.
- Quadratic solution $u_{\mathrm{e}} = bt^2 + Vt + I$ fulfills the ODE problem and the discrete equations with linear damping, but not for quadratic damping. A special discrete source term can allow u_{e} to also fulfill the discrete equations with quadratic damping.

Demo program

`vib.py` supports input via the command line:

```
Terminal> python vib.py --s 'sin(u)' --F '3*cos(4*t)' --c 0.03
```

This results in a moving window following the function on the screen.



Euler-Cromer formulation

We rewrite

$$mu'' + f(u') + s(u) = F(t), \quad u(0) = I,\ u'(0) = V,\ t \in (0, T]$$

as a first-order ODE system

$$u' = v$$
$$v' = m^{-1}\left(F(t) - f(v) - s(u)\right)$$

Staggered grid

- u is unknown at t_n: u^n
- v is unknown at $t_{n+1/2}$: $v^{n+\frac{1}{2}}$
- All derivatives are approximated by centered differences

$$
\begin{aligned}
&[D_t u = v]^{n-\frac{1}{2}} \\
&[D_t v = m^{-1}\left(F(t) - f(v) - s(u)\right)]^n
\end{aligned}
$$

Written out,

$$
\begin{aligned}
\frac{u^n - u^{n-1}}{\Delta t} &= v^{n-\frac{1}{2}} \\
\frac{v^{n+\frac{1}{2}} - v^{n-\frac{1}{2}}}{\Delta t} &= m^{-1}\left(F^n - f(v^n) - s(u^n)\right)
\end{aligned}
$$

Problem: $f(v^n)$

Linear damping

With $f(v) = bv$, we can use an arithmetic mean for bv^n a la Crank-Nicolson schemes.

$$
\begin{aligned}
u^n &= u^{n-1} + \Delta t v^{n-\frac{1}{2}}, \\
v^{n+\frac{1}{2}} &= \left(1 + \frac{b}{2m}\Delta t\right)^{-1}\left(v^{n-\frac{1}{2}} + \Delta t m^{-1}\left(F^n - \frac{1}{2}f(v^{n-\frac{1}{2}}) - s(u^n)\right)\right).
\end{aligned}
$$

Quadratic damping

With $f(v) = b|v|v$, we can use a geometric mean

$$
b|v^n|v^n \approx b|v^{n-\frac{1}{2}}|v^{n+\frac{1}{2}},
$$

resulting in

$$
\begin{aligned}
u^n &= u^{n-1} + \Delta t v^{n-\frac{1}{2}}, \\
v^{n+\frac{1}{2}} &= (1 + \frac{b}{m}|v^{n-\frac{1}{2}}|\Delta t)^{-1}\left(v^{n-\frac{1}{2}} + \Delta t m^{-1}\left(F^n - s(u^n)\right)\right).
\end{aligned}
$$

Initial conditions

$$
\begin{aligned}
u^0 &= I \\
v^{\frac{1}{2}} &= V - \frac{1}{2}\Delta t \omega^2 I
\end{aligned}
$$